Exploring NISC Architectures for Matrix Application

Sadhna K. Mishra[1], Arvind Rajawat[2], and R. P. Singh[2]

[1] MANIT/Comp. Science & Engg, Bhopal, India
Email: sadhnaguddy@rediffmail.com

[2] MANIT/Elect. & Communication, Bhopal, India
Email:{rajawata, singhrp}@manit.ac.in

***Abstract*— The paper presents the design of target NISC (No Instruction Set Computer) architecture for matrix application in a C based design flow. It starts with the implementation of a standard application program which generates customized designs using the NISC toolset. Further, it demonstrates and analyzes the compilation and simulation results of several matrix applications on a number of different available NISC architectures in terms of register and execution cycle-counts. Subsequently, a comparative analysis has been presented to explore the options to select the best set of architecture.**

***Index Terms*— NISC, ASIP, CISC, RISC, Datapath.**

I. INTRODUCTION

In the recent years, with increased complexities of embedded systems, the people involved in designing have been searching for a new alternative approach that could handle complexities and subsequently lead to dual target of achieving better design and to meet the strict design constraint such as size, timing, performance trade-offs, etc. During the past two decades, several design methodologies have emerged such as (i) Application-Specific Instruction-Set Processors (ASIP), (ii) High-level Synthesis (HLS), etc (figure 1). The concept of NISC (No- Instruction-Set-Computer) is a step forward in this endeavor. NISC concept offers an entirely new approach for design of custom processor.

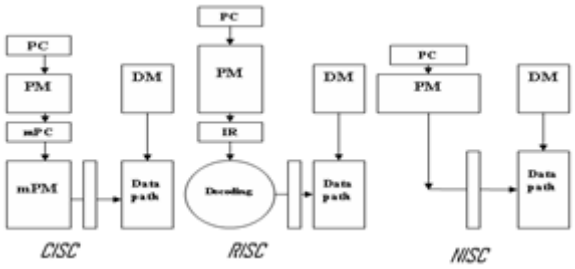

Figure1: Layout of CISC, RISC and NISC [15]

Basically, the NISC toolset is used primarily for (i) C-to-RTL synthesis and (ii) Embedded Custom Processor Design. The historical progression of the processor architecture could be broadly divided into three phases. During 1970s, it was CISC which was a popular choice. Given that the Program Memory (PM) was sluggish, the designers tried to improve performance by constructing complex instructions. While implementation, each complex instruction takes several clock cycles; with Datapath control words for each clock cycle are stored in a much faster Micro Program Memory (mPM). The very concept of micro programming allows for emulation of any instruction set and construction of specialized instruction, at the same time as speeding up the execution. However, on the down side the micro programming did not allow for efficient pipelining of the given Datapath [12] [14] [15] [16] [17]. During late 1980s, the RISC became popular and the very concept was to eliminate the complex instructions and the mPM. In RISC, all instructions are simple and they perform in one clock cycle allowing Datapath to be efficiently pipelined in 4-8 pipelined stages. Here, the mPM is replaced with decoding stage that follows the instruction fetch from PM. Given that instructions are simpler, a RISC wants approximately two instructions for each complex instruction and, consequently, the size of the PM is doubled. Nevertheless, the Fetch-Decode-Execute-Store pipeline of the whole processor improved the execution speed several times in comparison to its predecessor [12] [14] [16]. The present paper is based on NISC toolset, and the work compiles the different matrix applications results in relation to the design of applications on different available NISC and other custom architectures. In matrix application, the aim was to show the use of embedded processors in embedded control systems. These processors required performance in terms of basic mathematical abilities, bit manipulation etc. The work involved the generation of a set of Verilog codes of benchmark C codes and subsequently to explore different options to achieve the best set of results. The paper has been designed with various sections: section 2 concentrates on "NISC Architecture"; section 3 focuses on "NISC Methodology"; section 4 says about "Project Methodology"; section 5 explains the "Implementation", and finally the sections 6 summarizes the "Conclusion" of the study.

II.NISC ARCHITECTURE

As shown in figure 2, a typical NISC architecture is comprised of (i) *Control Pipelining* i.e. CW and Status register, (ii) *Datapath Pipelining* i.e. pipelined components or registers at input/output of components, and (iii) *Data Forwarding* i.e. the dotted connection lines from output of some components to input of some others. Here, the *control word* register (CW) the controls for both the datapath and the address generator (AG) of the controller, and the datapath section of CW contains the control values of all datapaths'

*Research Scholar, MANIT, Bhopal, India, 462051, E-mail: sadhnaguddy@rediffmail.com, Tel: (0755) 3296849.
**Assot Prof., Elect. & communication, MANIT Bhopal, India, 462051, E-mail: rajawata@manit.ac.in, Tel: (0755) 2670558.
*** Prof., Elect. & communication, MANIT Bhopal, India, 462051, E-mail: singhrp@manit.ac.in, Tel: (0755) 2671666.)

DOI: 01.IJCOM.02.02.157

components as well as a small constant fields [11, 15].

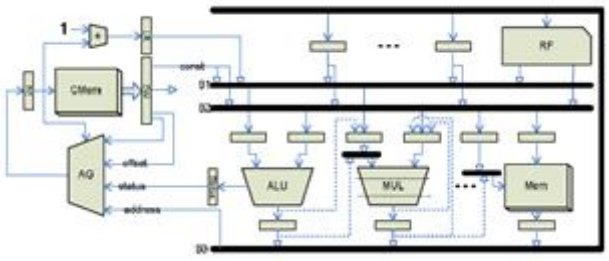

Figure 2: The NISC Processor [11, 15]

Simultaneously, the controller section of CW determines how the next PC address is calculated, and it provides a condition, a jump type either (i) direct or (ii) indirect, and an offset to the AG. For indirect jumps, AG calculates the target address by adding the offset and the current value of PC. At the same time, AG uses the value on its address port as target address. If the condition in CW and the *status* input of the AG are equal, then the calculated target address is loaded into PC, in other words it is incremented. Further, in NISC processor, there is a link register (LR) in the controller which stores the return address of a function call and the return address is usually the value of current PC plus one. In addition to standard components, the datapath could have pipelined and multi-cycle components such as *ALU*, *MUL* and *Mem* which are single-cycle, pipelined and multi-cycle components, respectively. There is no limitation on the connections of components in the datapath, however if the input of a component comes from multiple sources, a *Bus* or a *Multiplexer* is used to select the actual input. The buses are explicitly modeled and we assume one control bit per each writer to the bus. The multiplexers are implicit and we assume $\log^2 n$ control bits for n writers [11, 15].

III. NISC METHODOLOGY

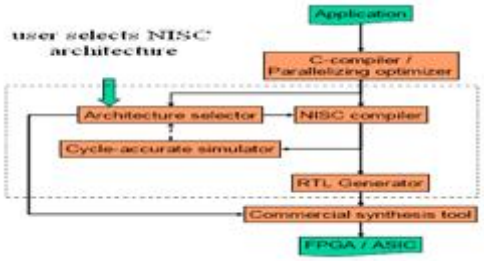

Figure 3: The Methodology of NISC [7]

NISC methodology is relatively simple and well defined to be used by application programmers. Here, the user specifies an application in C programming language and after studying the program structure, the user selects NISC architecture from the list or defines his/her own architecture by selecting components and their connectivity from a component list. Using the NISC compiler the C program is compiled for the given architecture. After compilation the result can be simulated and evaluated. Improvements can be made by changing original C program or selected architecture. At the end, RTL generator is used to generate Verilog RTL code for FPGA/ASIC implementation. A precise view is shown in figure 3 [5] [7].

IV. PROJECT METHODOLOGY

Project methodology is presented though two sets of experiments.

A. C- to –RTL synthesis

It presents the compilation and simulation results of several benchmarks of different categories on different NISC architectures. It was compiled and after running matrix applications on a set of generic NISC architectures, these architectures were as below:

GN_0, which was comprised of *No-Pipelined Datapath*, No data forwarding path, Single port memory, 2 Input bus, 1 Output bus, 1 Shared constant/offset port, 1 RF2x1 integer, 1 ALU, 1 Multiplier_ signed, 1 Display, 1 Comparator and 1 Converter;

GN_1, which was comprised of *No-Pipelined Datapath*, No data forwarding path, Single port memory, 2 input bus, 1 Output bus, 1 Shared constant/offset port, 1 RF2x1 integer, 1 ALU, 1 Multiplier_signed, 1 Display , 1 Comparator, 1 Converter, *1 Interrupt unit, 1 External output register*;

GN_2, which was comprised of *Pipelined Datapath* , No data forwarding path, Single port memory, 2 Input bus, 1 Output bus, 1 Shared constant/offset port, 1 RF2x1 integer, 1 ALU, 1 Multiplier_signed, 1 Display , 1 Comparator and 1 Converter;

GN_3, which was comprised of *Pipelined Datapath*, No data forwarding path, Single port memory, 2 Input bus, 1 Output bus, 1 Shared constant/offset port, 1 RF2x1 integer, 1 ALU, 1 Multiplier_signed, 1 Display , 1 Comparator, 1 Converter, *1 Interrupt unit, 1 External output register.*

GN_4, which was comprised of *Pipelined Datapath, Full data forwarding path*, Single port memory, 2 Input bus, 1 Output bus, 1 Shared constant/offset port, 1 RF2x1 integer, 1 ALU, 1 Multiplier_signed, 1 Display , 1 Comparator and 1 Converter.

GN_5, which was comprised of *Pipelined Datapath, Full data forwarding path*, Single port memory, 2 Input bus, 1 Output bus, 1 Shared constant/offset port, 1 RF2x1 integer, 1 ALU, 1 Multiplier_signed, 1 Display , 1 Comparator, 1 Converter, *1 Interrupt unit, 1 External output register.*

Finally, the NMIPS, which was comprised of 2 Input bus, 1 Output bus, RF2x1, 1 Shared constant/offset port, Single port memory, ALU, Multiplier_signed, Display. This datapath was as similar as possible to that of MIPS M4k [8].

B. Embedded Custom Processor Design

The methodology involved the *Design Embedded Custom Processors* for specifying the datapath and the custom functional units and then compiled the different applications and did analyze the aspects of suitability of a specific architecture for a particular class of application. The first NISC in this study was comprised of Single port memory, RF6x3, 2 ALUs, 2 Comparators, 2 Multiplier_signed, 2 Divider_signed, 2 Divider_unsigned, Display and 3 constants. The second NISC in this study contains Single

DOI: 01.IJCOM.02.02.157

port memory, RF6x3, 1 ALUs, 1 Comparators, 1 Multiplier_signed, 1 Divider_signed, 1 Divider_unsigned, Display and 3 constants. Finally, the NISC in this study was comprised of Single port memory, RF6x3, 1 ALUs, 1 Comparators, 1 Multiplier_signed, Display and 3 constants.

V. IMPLEMENTATION

The entire work was implemented though a set of application C program which was compiled and allowed run on a set of generic NISC architectures namely GN_0, GN_1, GN_2, GN_3, GN_4, GN_5, NMIPS, and finally the Design Embedded Custom Processor Custom Arch 1,Custom Arch 2,Custom Arch 3, which have been mentioned earlier. In fact, matrix multiplication is a binary operation that takes a pair of matrices, and produces another matrix. Matrix offers a concise way of representing linear transformations between vector space and matrix multiplication which corresponds to the composition of linear transformations. Resulting matrix agrees with the result of composition of the linear transformations represented by the two original matrices.
The product of an $m \times p$ matrix A with a $p \times n$ matrix B is an $m \times n$ matrix denoted AB whose entries are:

$$(AB)i, j = \sum_{k=1}^{p} AikBkj$$

where 1 d" *i* d" *m* is the row index and 1 d" *j* d" *n* is the column index.
The experimental outputs from these exercises are shown below in terms of register and execution cycle-counts of each matrix application.

A. C- to –RTL synthesis & Embedded Custom Processor Design Result Basis on RF Max

The results of RF Max value of the available NISC architecture and applications have been presented in table I. All the programs were scheduled with the help of five major parts such as _$jump ToStartupMain, NiscInterrupt, _$ NiscstartupMain, Application program (e.g. Matrix Multiplication, Matrix Multiplication unroll-1, Matrix Multiplication unroll-2) and NiscMain. Most of the registers were using the Application Programs. Computation was done in four steps mainly calculating the memory address, loading the values from data memory, operation perform and finally generating the results. Briefly, the results indicated that GN_4 and GN_5 (pipelined and data forwarding data path) had lowest units of *RF Max Index*. If we compare the value of MIPS M4K, the performance of GN_4 and GN_5 were good because Application Programs (e.g. Matrix Multiplication, Matrix Multiplication unroll-1, Matrix Multiplication unroll-2) had required less number of RF. So the RF values for GN_4 and GN_5 were good. Figure 4 is the graphical representation of table I.

TABLEI. RF MAX VALUES OF AVAILABLE ARCHITECTURES

	No pipelined Datapath		Pipelined Datapath and Data Forwarding path				Similar to MIPS M4K
	GN_0	GN_1	GN_2	GN_3	GN_4	GN_5	NMIPS
Matrix Multiplication [8] [8]	12	13	9	9	9	9	9
Matrix Multiplication unroll-1	11	11	10	10	9	10	10
Matrix Multiplication unroll-2	28	28	28	28	14	14	15

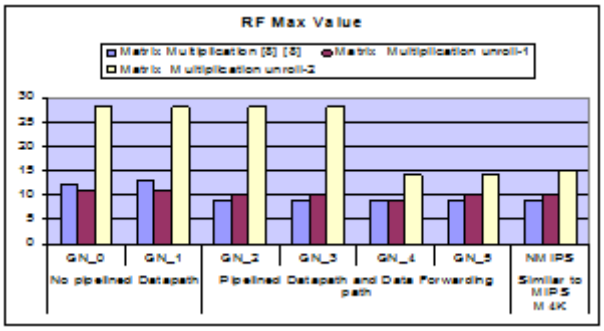

Figure 4. RF Max Value of Available NISC Architecture.

The outputs of RF Max value of the custom NISC architecture with matrix application have been presented in table II. As shown, all the Custom Architectures had no pipelined datapath so the results of custom architectures had been compared to GN_0 and GN_1 (both have no pipelined datapath). The result indicated that the Custom Arch 1, Custom Arch 2 and Custom Arch 3 in matrix multiplication [8] [8] had comparatively lower value of RF Max. The fourth row of the table II shows the results depicting that all the custom architecture required more number of RF where the GN_0 and GN_1were the most suitable for matrix multiplication unroll-1. Fifth row of the table shows that custom Architecture 1 was the best situation as compared to GN_0 and GN_1 given that it had less number of required RF for Application Program for this specific custom architecture. Figure 5 is the graphical depiction of table II.

TABLE II. RF MAX VALUE CUSTOM NISC ARCHITECTURE

	No pipelined Datapath Custom Arcitecture			No pipelined Datapath	
	Custom Arch 1	Custom Arch 2	Custom Arch 3	GN_0	GN_1
Matrix Multiplication [8] [8]	11	11	11	12	13
Matrix Multiplication unroll-1	26	14	14	11	11
Matrix Multiplication unroll-2	23	44	44	28	28

DOI: 01.IJCOM.02.02.157

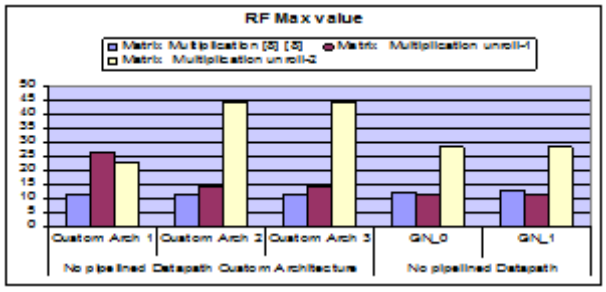

Figure 5. RF Max Value of Custom NISC Architecture

A. C- to –RTL synthesis & Embedded Custom Processor Design Result Basis on Cycle Count

Table III presents the result of cycle count of the available NISC architecture and matrix application. Briefly, the result shows that the GN_1, GN_2 (no pipeline) were suitable architectures of matrix multiplication [8][8], matrix multiplication unroll-1. As mentioned earlier, the programs were schedule in the following five parts _$jump ToStartupMain, NiscInterrupt, _$ NiscstartupMain, Application program and NiscMain. In GN_0 and GN_1 the part NiscInterrupt and NiscMain had required less number of cycle count, and thus it could had affected the total cycle count.

TABLE III.
CYCLE COUNT VALUE AVAILABLE ARCHITECTURE

	No pipelined Datapath		Pipelined Datapath and Data Forwarding path				Similar to MIPS M4K
	GN_0	GN_1	GN_2	GN_3	GN_4	GN_5	NMIPS
Matrix Multiplication [8] [8]	178	183	234	234	191	191	156
Matrix Multiplication unroll-1	204	204	265	265	205	208	174
Matrix Multiplication unroll-2	503	503	590	590	432	432	375

However, the GN_4, GN_5 (pipelined with data forwarding data path) were the suitable architectures of matrix multiplication unroll-2 and they had the lowest value of cycle count. Figure 6 is the graphical representation of table III.

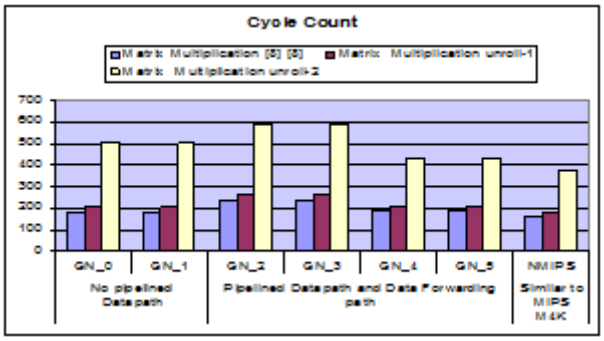

Figure 6. Cycle Count Value of Available NISC Architecture

The results of cycle count values of the custom NISC architecture with matrix multiplication have been presented in table IV.

TABLE IV.
CYCLE COUNT VALUE OF CUSTOM ARCHITECTURE

	No pipelined Datapath Custom Architecture			No pipelined Datapath	
	Custom Arch 1	Custom Arch 2	Custom Arch 3	GN_0	GN_1
Matrix Multiplication [8] [8]	122	126	126	178	183
Matrix Multiplication unroll-1	133	141	141	204	204
Matrix Multiplication unroll-2	279	291	291	503	503

Given, all the Custom Architectures had no pipeline datapaths so the results of these architectures have been compared to GN_0 and GN_1 (both are no pipelined). The first column of the table shows that Custom Arch 1 had the lowest units of cycle count so it was the best possible architecture of all *Matrix Application* on the basis of cycle count. Figure 5 is the graphical illustration of table IV.

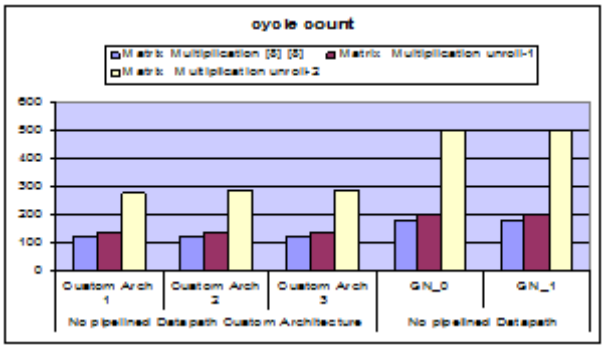

Figure 7. Cycle Count Value of Custom Architecture

VI. CONCLUSIONS

The research paper is an attempt to incorporate the design of matrix multiplication C code implementation and it presents a set of customized designs in relation to register and cycle count. It also shows and analyzes the simulation result related with several benchmarks on a number of different available *NISC architectures* and *Custom Architectures* in relation to register and cycle count. As mentioned above, the initial result indicated that the GN_4 and GN_5 architectures were relatively suitable for all applications in respect to register Max index. But when we focus on custom architectures in terms of matrix multiplication [8] [8] and matrix multiplication unroll-2, the Custom Arch1 was the best option. Further, if we consider matrix multiplication unroll-1, the GN_0 and GN_1 were the best options. Finally, if we compare the results in respect to cycle count, GN_0 was suitable for matrix multiplication [8] [8] and matrix multiplication unroll-1 but if we see the results of matrix multiplication unroll-2, the GN_4 and GN_5 were the best options. If we see in holistic terms, the results of custom architecture (Custom Arch 1) proved to be the best result for all the matrix applications.

DOI: 01.IJCOM.02.02.157

REFERENCES

[1] M. Reshadi, P. Mishra & N.Dutta, "Hybrid Compiled Simulation: An efficient technique for instruction- set architecture Simulation", *ACM Transactions on Embedded Computing Systems (TECS)*, April 2009.

[2] B.Gorjiara, M.Reshadi and D. Gajski, Merged Dictionary Code Compression for FPGA Implementation of Custom Microcoded PEs, *ACM Transactions on Reconfigurable Technology and Systems*, 2008.

[3] B. Gorjiara and D. Gajski, Automatic Architecture Refinement Techniques for Customizing Processing Elements, *Design Automation Conference (DAC)*, June 2008.

[4] M. Reshadi, B. Gorjara and D. Gajski, C-Based Design Flow: A Case Study on G.729A for Voice over Internet Protocol, *Design Automation Conference (DAC)*, pp. 72-75, May 2008.

[5] NISC Technology website: http://www.cecs.uci.edu/~nisc/

[6] J. Trajkovic & D. Gajski, "Automatic Data Path Generation from C code for Custom Processors", *International Embedded Systems Symposium*, May 2007.

[7] B. Gorjiara and D. Gajski, FPGA-friendly Code Compression for Horizontal Microcoded Custom IPs, *FPGA* , pp. 108-115, February 2007.

[8] B. Gorjara, M. Reshadi & D.Gajski, "Designing a Custom Architecture for DCT Using NISC Technology", *Asia and South Pacific Design Automation Conference (ASPDAC)*, Design Contest, January 2006.

[9] B.Gorjara, M. Reshadi & D.Gajski, "NISC Communication Interface", *Center for Embedded Computer Systems*, TR 06-05, March 2006.

[10] J. Trajkovic & D Gajski,"Communication Design for No Instruction Set Computer", *Center for Embedded Computer Systems*, TR 05-09, July 2005.

[11] B.Gorjara, M. Reshadi & D.Gajski, "NISC Technology and Preliminary Results", *Center for Embedded Computer Systems*, TR 05-11, August 2005.

[12] M. Reshadi & D.Gajski, "No-Instruction-Set-Computer (NISC) Technology", *Center for Embedded Computer Systems*, pp 1-21, 2005.

[13] M. Reshadi and D. Gajski, NISC Modeling and Simulation, *Center for Embedded Computer Systems*, TR 04-08, pp. 2-5, March 2004.

[14] M. Reshadi and D. Gajski, "NISC Application and Advantages", *Center for Embedded Computer Systems*, TR 04-08, pp. 2-5, March 2004.

[15] M. Reshadi and D. Gajski, NISC Modeling and Compilation, *Center for Embedded Computer Systems*, TR 04-33, pp 2-7, December 2004.

[16] D. Gajski, NISC: The Ultimate Reconfigurable Component, *Center for Embedded Computer Systems*, TR 03-28, pp. 2-8, October 2003.

[17] M. Morris Mano, *Computer system Architecture*, Prentice Hall, India, 2003,ch.8, pp.282-285.

DOI: 01.IJCOM.02.02.157